Groupement de clés efficace pour un équilibrage de charge quasi-optimal dans les systèmes de traitement de flux

Nicoló Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, Bruno Sericola

Groupement de clés efficace pour un équilibrage de charge quasi-optimal dans les systèmes de traitement de flux

Nicoló Rivetti[1,2], Leonardo Querzoni[2], Emmanuelle Anceaume[3], Yann Busnel[4,5] et Bruno Sericola[5]

[1]*LINA / Université de Nantes, France*
[2]*Sapienza University of Rome*
[3]*IRISA / CNRS, Rennes, France*
[4]*Crest / Ensai, Rennes, France*
[5]*Inria Rennes – Bretagne Atlantique, France*

Le groupement de clés est une technique utilisée dans les plateformes de traitement de flux pour simplifier le déploiement parallèle d'opérateurs à états. Cette méthode consiste à subdiviser le flux de tuples en plusieurs sous-flux disjoints, au regard d'une clef définie pour chaque tuple. Chaque opérateur recevant l'un de ces sous-flux est donc assuré de recevoir tous les tuples contenant une clé spécifique. Une solution classique est de grouper les clés en utilisant des fonctions de hachage, induisant potentiellement un déséquilibre de charge des opérateurs en cas de distribution biaisée des clefs du flux d'entrée. Nous présentons une solution permettant d'obtenir une répartition de la charge proche de l'optimal, quelque soit la distribution. Cette solution repose sur le fait que l'équilibre est majoritairement influencé par les clés les plus fréquentes. En identifiant ces valeurs dominantes et en les groupant judicieusement avec des clés plus rares, les sous-flux ainsi obtenus permettent d'atteindre un équilibrage de charge quasi-optimal. Nous présentons une analyse théorique des bornes de qualité obtenu avec notre algorithme et illustrons son impact sur des applications de traitement de flux, via des simulations intensives et un prototype opérationnel.

Mots-clefs : Traitement de flux; Groupement de clé; Equilibrage de charge; Algorithme d'approximation probabiliste; Evaluation de performance

Travaux co-financés par le projet ANR SocioPlug (ANR-13-INFR-0003) et le projet DeSceNt du Labex CominLabs (ANR-10-LABX-07-01).

1 Introduction

Stream processing systems are today gaining momentum as a tool to perform analytics on continuous data streams. A stream processing application is commonly modeled as a direct acyclic graph where data operators, represented by nodes, are interconnected by streams of tuples containing data to be analyzed, the directed edges. Scalability is usually attained at the deployment phase where each data operator can be parallelized using multiple instances, each of which will handle a subset of the tuples conveyed by the operator's ingoing stream. Balancing the load among the instances of a parallel operator is important as it yields to better resource utilization and thus larger throughputs and reduced tuple processing latencies.

How tuples pertaining to a single stream are partitioned among the set of parallel instances of a target operator strongly depends on the application logic implemented by the operator itself. When the target operator is stateful (assignment based on the specific value of data contained in the tuple itself; *key* or *field grouping*) things get more complex as its state must be maintained continuously synchronized among its instances, with possibly severe performance degradation at runtime; a well-known workaround to this problem consists in partitioning the operator state and let each instance work on the subset of the input stream containing all and only the tuples which will affect its state partition, simplifying the work of developing parallelizable stateful operators.

The downside of using key grouping is that it may induce noticeable imbalances in the load experienced by the target operator whenever the data value distribution is skewed, a common case for many application scenarios. This is usually the case with implementations based on hash functions: the hashed data is mapped, for example using modulo, to a target instance. A solution could lie in defining an explicit one-to-one mapping between input values and available target instances that could take into account the skewed frequency distribution and thus uniformly spread the load; this solution is considered impractical as it requires to have precise knowledge on the input value distribution (usually not known) and imposes, at runtime, a memory footprint that is proportional to the huge number of possible values in the input domain.

In this paper we propose a new key grouping technique called Distribution-aware Key Grouping (DKG) targeted toward applications working on input streams characterized by a skewed value distribution. DKG is based on the observation that when the values used to perform the grouping have skewed frequencies, *e.g.,* they can be approximated with a Zipfian distribution, the few most frequent values (the *heavy hitters*) drive the load distribution, while the remaining largest fraction of the values (the *sparse items*) appear so rarely in the stream that the relative impact of each of them on the global load balance is negligible. However, when considered in groups sparse items should be handled with care. After an initial training phase, whose length can be tuned, the final mapping is obtained by running a greedy multiprocessor scheduling algorithm that takes as input the heavy hitters with their estimated frequencies and the groups of sparse items with their frequencies and outputs a one-to-one mapping of these elements to the available target instances. The final result is a mapping that is fine-grained for heavy hitters, that must be carefully placed on the available target instances to avoid imbalance, and coarse-grained for the sparse items whose impact on load is significant only when they are considered in large batches. DKG is a practical solution as it uses efficient data streaming algorithms to estimate with a bounded error the frequency of the heavy hitters, and has a small and constant memory footprint that can be tuned by system administrators, thus overcoming the two main limitations of standard one-to-one mappings.

We show, through a theoretical analysis, that DKG provides on average near-optimal mappings using sub-linear space in the number of tuples read from the input stream in the learning phase and the support (value domain) of the tuples. We also extensively tested DKG both in a simulated environment with synthetic datasets and on a prototype implementation based on Apache Storm (`http://storm.apache.org`) running with real data. The experiments show that DKG outperforms standard hashing solutions when run over streams with slightly- to strongly-skewed value distributions. In particular, DKG is able to deliver very stable performance that do not vary strongly with the input, and that are always close to an approximate optimum that is obtainable only with full information on the input distribution. Due to space constraints, proofs and main experimental results are available in the companion paper [RQA+15], which the interested reader is invited to consult.

2 Distribution-aware Key Grouping

In this section we present our solution consisting of a three-phase algorithm. (i) In the first phase the algorithm becomes aware of the stream distribution (*learning* phase). (ii) The following phase builds a global mapping function taking into account the previously gathered knowledge of the stream (*build* phase). (iii) The last phase uses the built global mapping function to perform key grouping with near-optimal balancing (*deployment* phase). As previously motivated, we cannot afford to store the entire input data stream in order to analyze it and cannot make multiple passes over it to keep pace with the rate of the stream. As such we rely on data streaming algorithms, which have shown their highly desirable properties in data intensive applications to compute different kind of basic statistics.

Data Stream Model and Heavy Hitters A stream is a sequence of elements $\langle t_1, t_2, \ldots, t_m \rangle$ called tuples or items, which are drawn from a large universe $[n] = \{1, \ldots, n\}$. In the following, the size (or length) of the stream is denoted by m. Notice that in this work we do not consider m as the size of the whole stream, but as the size of the learning set, *i.e.,* how long the first phase lasts. In the following we denote by p_i the unknown probability of occurrence of item i in the stream and with f_i the unknown frequency of item i, *i.e.,* the mean number of occurrences of i in the stream of size m. An item i of a stream is called *heavy hitter* if the empirical probability p_i with which item i appears in the stream satisfies $p_i \geq \Theta$ for some given

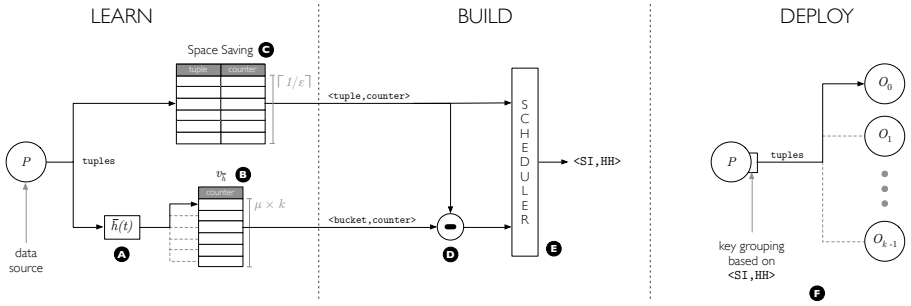


Figure 1: DKG architecture and working phases.

threshold $0 < \Theta \leq 1$. In the following we call *sparse items* the items of the stream that are not heavy hitters.

Greedy scheduling algorithm We adapt the *Multiprocessor Scheduling* problem to our setting. More formally we have (i) a set of buckets $b_i \in \mathcal{B}$, each of them with an associated frequency f_i, and (ii) a set of instances $r_j \in \mathcal{R}$ ($|\mathcal{R}| = k$), each of them with an associated load L_j. The load L_j is the sum of the frequencies of the buckets assigned to instance r_j. We want to associate each bucket $b_j \in \mathcal{B}$, minimizing the maximum load on the instances: $\max_{j=1,\ldots,k}(L_j)$. To solve efficiently this problem, known to be NP-hard, we make use of the classic Least Processing Time First (LPTF) approximation algorithm run over small inputs. The algorithm (referred as `scheduling` algorithm in the following) assigns the bucket $b_i \in \mathcal{B}$ with the largest frequency f_i to the instance r_j with the lowest load L_j, then removes b_i from $\mathcal{B}$ and repeats until $\mathcal{B}$ is empty. This algorithm provides then a $(\frac{4}{3} - \frac{1}{3k})$-approximation of the optimal mapping [Gra69].

DKG design As previously mentioned, the unbalancing in key grouping is mainly due to the skewness of the input stream distribution. For instance, let i and j be the stream heavy hitters, most likely a good mapping should keep them separated. In addition, if $f_i > m/k$, the hash function should isolate i on an instance. However, a randomly chosen hash function will almost certainly assign other (sparse) items with i. Even worse, the hash function may end up putting i and j together. To cope with these issues, DKG becomes aware of the stream distribution through a learning phase. It is then able to build a global mapping function that avoids pathological configurations and that achieves close to optimal balancing.

DKG (Figure 1.A) chooses a hash function $\bar{h} : \{1,\ldots,n\} \rightarrow \{1,\ldots,k\mu\}$ randomly from a 2-universal hash functions family, where μ is a user defined parameter. Increasing the co-domain size reduces the collision probability, thus enhances the odds of getting a good assignment. Having more buckets (elements of $\bar{h}$ co-domain) than instances, we can map buckets to instances minimizing the unbalancing. More in details, DKG feeds to the previously presented `scheduling` algorithm the buckets of $\bar{h}$ with their frequencies as the set $\mathcal{B}$ and the number of instances k. To track the frequencies of $\bar{h}$'s buckets, DKG keeps an array $v_{\bar{h}}$ of size $k\mu$. When receiving tuple t, DKG increments the cell associated through $\bar{h}$ with t (Figure 1.B). In other words $v_{\bar{h}}$ represents how $\bar{h}$ maps the stream tuples to its own buckets.

While this mechanism improves the balancing, it still does not deterministically guarantee that heavy hitters are correctly handled. If the buckets have in average the same load, the `scheduling` algorithm should be able to produce a good mapping. To approximate this ideal configuration we have to remove all the heavy hitters. As such, DKG uses the `SS` algorithm [MAEA05] to detect heavy hitters and manage them ad-hoc. To match the required detection and estimation precisions, the `SS` algorithm monitors $1/\varepsilon$ distinct keys, where $\varepsilon < \Theta$ (Θ is the relative frequency of heavy hitters and is a user defined parameter). When receiving tuple t, DKG updates the `Space Saving` algorithm instance (Figure 1.C). At the end of the learning phase, the `SS` algorithm will maintain the most frequent values of t and their estimated frequencies. Then (Figure 1.D), DKG removes the frequency count of each heavy hitter from $v_{\bar{h}}$. Finally (Figure 1.E), it feeds to the `scheduling` algorithm the $v_{\bar{h}}$ array and the heavy hitters from the `SS` algorithm, with the frequencies of both, as the set of bucket $\mathcal{B}$. The `scheduling` algorithm will return an approximation of the optimal mapping from $\bar{h}$ buckets and detected heavy hitters to instances.

When receiving tuple t in the deployment phase (Figure 1.F), the resulting global mapping function: (i) checks if t is a frequent item and returns the associated instance. (ii) Otherwise it computes the hash of t: $\bar{h}(t)$, and returns the instance associated with the resulting bucket.

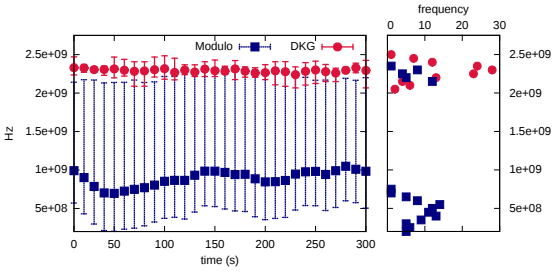
Theorem 2.1 (Time complexity of DKG) *The time complexity of DKG is $O(\log 1/\varepsilon)$ per update in the learning phase, $O((k\mu+1/\varepsilon)\log(k\mu+1/\varepsilon))$ to build the global mapping function, and $O(1)$ to return the instance associated with a tuple.*

Theorem 2.2 (Memory requirement of DKG) *The space complexity of DKG is $O((k\mu+1/\varepsilon)\log m+\log n)$ bits in the learning phase and to build the global mapping function. To store the global mapping function, DKG requires $O((k\mu+1/\Theta)\log n)$ bits.*

Theorem 2.3 (Accuracy of DKG) *DKG provides an $(1+\Theta)$-optimal mapping for key grouping using $O(k\mu\log m+\log n)$ bits of memory in the learning phase and $O(k\mu\log n)$ bits of memory to store the global mapping function, where $\mu \geq 1/\Theta \geq k$*

3 Experiment results

We also extensively tested DKG both in a simulated environment with synthetic datasets and on a prototype implementation running with real data. Due to space constraints, the extensive simulation results are available in the companion paper [RQA+15]. To evaluate the impact of DKG on real applications we implemented it[†] as a custom grouping function within the Apache Storm framework. The use case for our tests is a partial implementation of the third query from the DEBS 2013 Grand Challenge. The goal is to compute, over four different time windows and grid granularities, how long each of the monitored players is in each grid cell. Figure 2(a) shows the CPU usage (Hz) over time (300 seconds of execution) for DKG and Modulo (the default Apache Storm key grouping implementation) with the *DEBS trace* in the test topology with $k = 4$ instances. The CPU usage was measured as the average number of hertz consumed by each instance every 10 seconds. Plotted values are the mean, max and min CPU usage on the 4 instances. The mean CPU usage for DKG is close to the maximum CPU usage for Modulo, while the mean CPU usage for Modulo is much smaller. This was expected as there is a large resource under-utilization with Modulo. Figure 2(b) shows the CPU usage's distribution for the same time-slice. In other words the data point x-axis value represents how many times an instance has reached this CPU usage (Hz). Notice that the values are rounded to the nearest multiple of 5×10^7. This plot confirms that the CPU usage for DKG's replicas is concentrated between 2×10^9 and 2.5×10^9 Hz, *i.e.,* all instances are well balanced with DKG. On the other hand Modulo does hit this interval, but most of the data points are close to 5×10^8 Hz. The key grouping provided by DKG loads evenly all available instances. Conversely, with Modulo some instance (in particular 3) are underused, leaving most of the load on fewer instances (in particular 1). This improvement in resource usage translates directly into a larger throughput and reduced execution time; in particular, in our experiments, DKG delivered $2\times$ the throughput of Modulo for $k \in \{4,5,6,8,10\}$. The length of the learning phase can have a large impact on the algorithm performances and the applicability of the solution, however our tests show that DKG performances converge after less than $1,000$ tuples.



(a) Timeline (b) Distribution

Figure 2: CPU usage (Hz) for 300s of execution ($\Theta = 0.1$, $\varepsilon = 0.05$, $\mu = 2$ and $k = 4$)

References

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. on Applied Math.*, 17(2), 1969.

[MAEA05] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. of the 10th Intl Conf. on Database Theory (ICDT)*, 2005.

[RQA+15] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proc. of the 9th ACM Intl Conference on Distributed Event-Based Systems (DEBS 2015)*, Oslo, Norway, June 2015.

[†] The implementation's code is available at the following repository: `http://github.com/rivetti/dkg_storm`